IPSJ Transactions on System LSI Design Methodology Vol. 3 118–129 (Feb. 2010)

Regular Paper

Programmable Architectures and Design Methods for Two-Variable Numeric Function Generators ⋆1

Shinobu Nagayama,†1 Tsutomu Sasao†2
and Jon T. Butler†3

This paper proposes programmable architectures and design methods for numeric function generators (NFGs) of two-variable functions. To realize a two-variable function in hardware, we partition a given domain of the function into segments, and approximate the function by a polynomial in each segment. This paper introduces two planar segmentation algorithms that efficiently partition a domain of a two-variable function. This paper also introduces a design method for symmetric two-variable functions (i.e. $f(X,Y) = f(Y,X)$). This method can reduce the memory size needed for symmetric functions by nearly half with small speed penalty. The proposed architectures allow a systematic design of various two-variable functions. We compare our approach with one based on a one-variable NFG. FPGA implementation results show that, for a complicated function, our NFG achieves 57% of memory size and 60% of delay time of a circuit designed based on a one-variable NFG.

1. Introduction

The ability to compute numeric functions at a high speed is important in many applications [12], including 3D computer graphics, hardware accelerators for technical computing packages, direct digital frequency synthesizers [4], and digital signal processing. Various design methods for numeric function generators (NFGs) have been devised for numeric functions on one variable [5),10),14),15),18)–20)]. Only a few methods exist for multi-variable functions (e.g., $\sqrt{X^2+Y^2+Z^2}$ and $\arctan(X/Y)$) [6),7),22)]. However, these methods are function-specific; different functions require different methods. As far as we know, no systematic design method exists for generic multi-variable functions.

A straightforward design method for arbitrary multi-variable function is to use a single memory in which the address is a combination of values of variables and the content of that address is the corresponding value of function. This method produces a fast implementation, but requires a 2^{mn}-word memory to implement an m-variable function with n bits for each variable. Thus, unlike one-variable functions, even for a computation with a small number of bits, this method is impractical because of large memory size.

To produce a practical implementation, multi-variable functions are often designed in a conventional (trivial) manner that uses a combination of one-variable function generators, multipliers, and adders [6),7)]. For example, the function $\sqrt{X^2+Y^2+Z^2}$ can be realized using three circuits, each realizing a^2, two adders, and a square root circuit. This design method may require small memory size. However, depending on the function implemented, it can produce a slow implementation because of long path delays. Also, such circuits make error analysis harder. That is, guaranteeing output accuracy becomes harder. Also, there are many multi-variable functions that cannot be decomposed into one-variable functions, such as probability distributions that are functions of the random variable and a parameter, like variance.

This paper proposes a systematic design method for two-variable functions. Since our design method is based on a piecewise polynomial approximation, architectures are simple even for complicated functions. To approximate a given function using piecewise polynomials, we introduce two planar segmentation algorithms that efficiently partition a given domain of a two-variable function. We also introduce programmable architectures that can realize a wide range of two-variable functions.

The rest of this paper is organized as follows: Section 2 introduces the number representation and the decision diagrams used in this paper. Section 3 presents two planar segmentation algorithms and a polynomial approximation method using bilinear interpolation. Section 4 presents programmable architectures for two-variable functions. Section 5 presents an architecture and a design method for symmetric two-variable functions. Section 6 evaluates performance of our segmentation algorithms and architectures for two-variable functions. And, Sec-

†1 Hiroshima City University
†2 Kyushu Institute of Technology
†3 Naval Postgraduate School
⋆1 This paper is an extension of two papers 16), 17).

tion 7 concludes the paper. An error analysis for our NFGs is omitted because it is the almost same as Refs. 14), 19).

2. Preliminaries

2.1 Number Representation and Errors

Definition 1 *A **numeric function generator (NFG)** is a logic circuit that computes approximated values for a numeric (real) function within some given acceptable error ε. A **one-variable NFG** is a logic circuit for a one-variable numeric function $f(X)$, whose input is X, and output is an approximated value for $f(X)$. A **two-variable NFG** is a logic circuit for a two-variable numeric function $f(X,Y)$, whose inputs are X and Y, and output is an approximated value for $f(X,Y)$.*

Definition 2 *A value X represented by the **binary fixed-point representation** is denoted by*

$$X = (x_{l-1}\ x_{l-2}\ \ldots\ x_1\ x_0.\ x_{-1}\ x_{-2}\ \ldots\ x_{-m}),$$

*where $x_i \in \{0,1\}$, l is the number of bits in the integer part, and m is the number of bits in the fractional part. Each bit x_i contributes $2^i x_i$ to the **value** of X, except x_{l-1}, which contributes $-2^{l-1}x_{l-1}$. That is, the fixed-point representation is in two's complement.*

Definition 3 ***Error** is the absolute difference between the exact value and the value produced by the hardware. **Acceptable error** is the maximum error that an NFG may assume; it is usually a specification to be satisfied by the hardware. **Approximation error** is the error caused by a function approximation. **Acceptable approximation error** is the maximum approximation error that a function approximation may assume. **Rounding error** is the error caused by a binary fixed-point representation.*

Definition 4 ***Accuracy** is the number of bits in the fractional part of a binary fixed-point representation. **m-bit accuracy** specifies that m bits are used to represent the fractional part of the number. When the maximum error is 2^{-m}, the accuracy is no greater than 1 **unit in the last place (ULP)**[12]. In this paper, an **m-bit accuracy NFG** is an NFG with an m-bit fractional part of the inputs, an m-bit fractional part of the output, and a 1 ULP error.*

2.2 Decision Diagrams

The proposed design uses binary decision diagrams.

Definition 5 *A binary decision diagram (BDD)[2),11)] is a rooted directed acyclic graph (DAG) representing a logic function. The BDD is obtained by recursively applying the Shannon expansion $f = \overline{x_i} f_0 + x_i f_1$ to the logic function, where f, f_0, and f_1 are represented by nodes. There are two types of nodes, terminal nodes that are labeled by the two function values, 0 and 1, and non-terminal nodes that are labeled by variable names. Each non-terminal node has two unweighted outgoing edges labeled 0 and 1, corresponding to the value of the node's variable. The terminal nodes have no outgoing edges. We consider only ordered BDDs, where the order of the variables is the same for every path from the root node to a terminal node. We consider only reduced BDDs, where identical subtrees are combined into a single tree.*

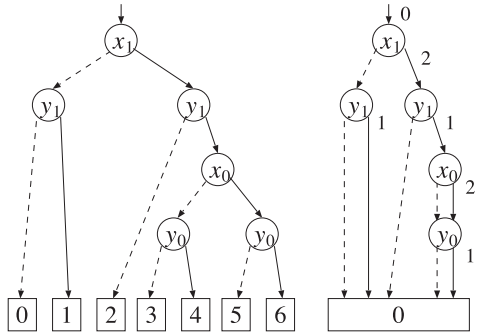
Definition 6 *A **multi-terminal BDD (MTBDD)**[3] is an extension of a BDD, that represents an integer-valued function: $\{0,1\}^n \to S \subseteq Z$, where S is a finite subset of the set Z of integers. In an MTBDD, the terminal nodes are labeled by values of S.*

Definition 7 *An **edge-valued BDD (EVBDD)**[8),9)] is also an extension of a BDD, that represents an integer-valued function. The EVBDD is obtained by repeatedly applying the expansion $f = \overline{x_i} f_0 + x_i(f_1' + \alpha)$ to the integer-valued function, where $f_1 = f_1' + \alpha$, and α is the constant term of f_1. In an EVBDD, all 1-edges have an integer weight and all 0-edges have weight 0. There is only one terminal node; it is labeled 0. The incoming edge into the root node can have a non-zero weight. A non-zero weight α on the incoming edge of the root node adds α to all sums associated with all paths from the root to the terminal node of the EVBDD. Indeed, it occurs when the EVBDD is a sub-EVBDD to a larger EVBDD.*

Example 1 **Figure 1** (b) *and* (c) *show an MTBDD and an EVBDD for the integer-valued function f defined by Fig. 1 (a). In Fig. 1 (b) and (c), dashed lines and solid lines denote 0-edges and 1-edges, respectively. Note the non-zero weights on 1-edges of the EVBDD. In the MTBDD, terminal nodes represent function values. Thus, to evaluate the function, we traverse the MTBDD from the root node to a terminal node according to the input values, and obtain the function*

x_1	y_1	x_0	y_0	f	x_1	y_1	x_0	y_0	f
0	0	0	0	0	1	0	0	0	2
0	0	0	1	0	1	0	0	1	2
0	0	1	0	0	1	0	1	0	2
0	0	1	1	0	1	0	1	1	2
0	1	0	0	1	1	1	0	0	3
0	1	0	1	1	1	1	0	1	4
0	1	1	0	1	1	1	1	0	5
0	1	1	1	1	1	1	1	1	6

(a) Function table.



(b) MTBDD

(c) EVBDD

Fig. 1 MTBDD and EVBDD for an integer-valued function.

value (an integer) from the terminal node. On the other hand, in the EVBDD, we obtain the function value by summing the weights of the edges traversed from the root node to the terminal node. *(End of Example)*

3. Piecewise Polynomial Approximation Based on Planar Segmentation

3.1 Planar Segmentation Problem

To approximate a given two-variable function by piecewise polynomials, we partition a given domain of the function into segments, and approximate the function by a polynomial in each segment. By narrowing segments, and thus increasing the number of segments, we can decrease the approximation error to the desired value. In this case, the memory size and speed of an NFG are strongly dependent on segmentation of domain. Thus, to design fast and compact NFGs, we need to solve the following segmentation problem: Given a two-variable function, its domain, and acceptable approximation error, find an optimum segmentation. To find an optimum segmentation, we consider the following:

(1) number of words in the *coefficients memory*, which is the number of segments, and

(2) complexity of hardware to realize segmentation, called the *segment index encoder*, which maps values of X and Y to a segment number.

Fewer segments are preferred because the number of segments directly affects the size of the coefficients memory of the NFG. But, the complexity of the segment index encoder is important as well. Even if the number of segments is minimum, a large NFG is produced if the segment index encoder is very large.

For one-variable functions, since the domain is formed in one-dimension (line), any segmentation can be realized compactly. Thus, we considered only the number of segments to find an optimum segmentation [14),19)]. On the other hand, for two-variable functions, since the domain is formed in two-dimensions (plane), the segment index encoders tend to be much more complex than for one-variable functions. Thus, to find the optimum design of two-variable NFGs, it is necessary to carefully consider the complexity of the segment index encoder.

For one-variable functions, we have proposed linear segmentation algorithms [14),19)] to find an optimum segmentation of a linear domain (an approximation with the fewest segments) efficiently. However, for two-variable functions, a planar segmentation algorithm is now required to find an optimum segmentation of a planar domain. In planar segmentations, we have a higher degree of freedom, and thus, finding an optimum segmentation becomes much more difficult than in linear segmentation. Because many segments may be involved in a practical design, the time needed to find an optimum segmentation can be very long. To produce an efficient planar segmentation in a short computation time, we focus on heuristic planar segmentation algorithms. The next subsection presents two heuristic planar segmentation algorithms that produce an efficient planar segmentation by regularly partitioning a given domain using squares.

3.2 Planar Segmentation Algorithms

We first present a *recursive planar segmentation algorithm* to reduce the hardware complexity of both the coefficients memory (the number of segments) and the segment index encoder. **Figure 2** shows this algorithm. Inputs of the algorithm are a numeric function $f(X,Y)$, a domain $\{[X_b,X_e),[Y_b,Y_e)\}$ for X and Y, an accuracy m_{in} of X and Y, and an acceptable approximation error ε_a. This algorithm begins by computing an approximate polynomial $g(X,Y)$. This is an initial approximation. If that approximation error ε is larger than the given acceptable error ε_a, then the domain is partitioned into four equal-sized square segments. For each segment, an approximate polynomial is computed again. The

Input:	Numeric function $f(X,Y)$, domain $\{[X_b,X_e),[Y_b,Y_e)\}$ for X and Y, accuracy m_{in} of X and Y, and acceptable approximation error ε_a. X and Y are represented in the same number of bits.
Output:	Segments $\{[X_b,P_0),[Y_b,Q_0)\},\{[X_b,P_0),[Q_0,Q_1)\}\ldots,\{[P_{r-1},X_e),[Q_{r-1},Y_e)\}$, and correction values $v_0,v_1,\ldots,v_{k-1}$.
Step:	
1.	For $\{[X_b,X_e),[Y_b,Y_e)\}$, compute an approximate polynomial $g(X,Y)$.
2.	Compute the maximum positive error $max_{fg}=\max\{f(X,Y)-g(X,Y)\}$.
3.	Compute the maximum negative error $min_{fg}=\min\{f(X,Y)-g(X,Y)\}$.
4.	Compute approximation error $\varepsilon=(max_{fg}-min_{fg})/2$ and correction values $v=(max_{fg}+min_{fg})/2$.
5.	If $\varepsilon<\varepsilon_a$ or $(X_e-X_b)\le 2^{-m_{in}}$, then stop.
6.	Else, partition $\{[X_b,X_e),[Y_b,Y_e)\}$ into four segments $\{[X_b,P),[Y_b,Q)\}$, $\{[X_b,P),[Q,Y_e)\}$, $\{[P,X_e),[Y_b,Q)\}$, and $\{[P,X_e),[Q,Y_e)\}$, where $P=(X_b+X_e)/2$ and $Q=(Y_b+Y_e)/2$.
7.	Repeat Steps 1, $2,\ldots,6$ for each new segment recursively, until the maximum approximation errors are smaller than ε_a in all segments.

Fig. 2 Recursive planar segmentation algorithm.

same process is recursively repeated until all segments have approximation errors smaller than ε_a. Note that this algorithm creates a segment of size $w_i \times w_i$, where $w_i = 2^{h_i} \times 2^{-m_{in}}$ and h_i is an integer. That is, all the segmentation points P_i and Q_i are restricted to values such that the least significant h_i bits are 0 (i.e., $P_i = (\ldots\ p_{-j+1}\ p_{-j}\ 00\ \ldots\ 0)$, where $j = m_{in} - h_i$). This restriction contributes to reduce the complexity of the segment index encoder.

Next, we present the *planar uniform segmentation algorithm*. Since the recursive planar segmentation algorithm produces *non-uniform* segmentation, a segment index encoder is needed to compute a segment number from values of X and Y. However, in a *uniform* segmentation where the number of segments is a power of 2, a segment index encoder is not necessary because a segment number is obtained by the most significant bits of X and Y (see **Fig. 3** (b)). This eliminates the delay of the segment index encoder, and produces fast NFGs. To produce a uniform segmentation, we begin by finding the smallest square segment needed to achieve the acceptable approximation error using the recursive segmentation algorithm shown in Fig. 2. Then, we partition a given domain into square segments all with the same size as the smallest segment.

3.3 Approximation Using Bilinear Interpolation Polynomials

For $g(X,Y)$ in Fig. 2, we can use any approximating polynomial. In general, higher-order polynomials require fewer segments. However, for multi-variable functions, using higher-order polynomials is not always effective in reducing the memory size of NFGs. This is because, for multi-variable polynomials, higher polynomial order requires many more polynomial coefficients. Also, higher-order polynomials produce slower NFGs. Thus, for polynomial approximation methods, reducing memory size with a small speed penalty is a key issue. To accomplish this, we use the bilinear interpolation polynomials [21].

Bilinear interpolation is an extension of linear interpolation. It interpolates two-variable functions $f(X,Y)$ using four points. In Fig. 2, to interpolate $f(X,Y)$ in each segment $\{[B_x,E_x),[B_y,E_y)\}$, we use four corner points of the segment: (B_x,B_y), (B_x,E_y), (E_x,B_y), and (E_x,E_y). Let $f_{bb}=f(B_x,B_y)$, $f_{be}=f(B_x,E_y)$, $f_{eb}=f(E_x,B_y)$, and $f_{ee}=f(E_x,E_y)$. Then, the bilinear interpolation $g(X,Y)$ is given by:

$$g(X,Y)=\frac{f_{bb}\times(E_x-X)\times(E_y-Y)+f_{eb}\times(X-B_x)\times(E_y-Y)}{(E_x-B_x)\times(E_y-B_y)} + \frac{f_{be}\times(E_x-X)\times(Y-B_y)+f_{ee}\times(X-B_x)\times(Y-B_y)}{(E_x-B_x)\times(E_y-B_y)}.$$

By expanding and rearranging this, we obtain the following form:

$$g(X,Y)=C_{xy}XY+C_xX+C_yY+C_0,$$

where

$$C_{xy}=\frac{f_{bb}-f_{eb}-f_{be}+f_{ee}}{(E_x-B_x)(E_y-B_y)},$$
$$C_x=\frac{-f_{bb}E_y+f_{eb}E_y+f_{be}B_y-f_{ee}B_y}{(E_x-B_x)(E_y-B_y)},$$
$$C_y=\frac{-f_{bb}E_x+f_{eb}B_x+f_{be}E_x-f_{ee}B_x}{(E_x-B_x)(E_y-B_y)},\ \text{and}$$
$$C_0=\frac{f_{bb}E_xE_y-f_{eb}B_xE_y-f_{be}E_xB_y+f_{ee}B_xB_y}{(E_x-B_x)(E_y-B_y)}.$$

To reduce the approximation error, the maximum positive error max_{fg} and the maximum negative error min_{fg} are equalized by a vertical shift of $g(X,Y)$ with a correction value $v=(max_{fg}+min_{fg})/2$. Thus, the approximation error is $(max_{fg}-min_{fg})/2$, and the approximating polynomial is $g(X,Y)+v$.

For each segment $\{[B_x,E_x),[B_y,E_y)\}$, since $B_x \le X < E_x$ and $B_y \le Y <$

E_y hold, we can offset X and Y by B_x and B_y to compute the approximating polynomial $g(X,Y)+v$. By using the offset inputs $(X-B_x)$ and $(Y-B_y)$ instead of X and Y, we reduce the size of multipliers needed to compute $g(X,Y)+v$. By substituting $X-B_x+B_x$ and $Y-B_y+B_y$ for X and Y respectively, we transform $g(X,Y)+v$ as follows:

$$\begin{aligned} g(X,Y)+v &= C_{xy}(X-B_x+B_x)(Y-B_y+B_y)+C_x(X-B_x+B_x) \\ &\quad +C_y(Y-B_y+B_y)+C_0+v \\ &= C_{xy}(X-B_x)(Y-B_y)+(C_x+C_{xy}B_y)(X-B_x) \\ &\quad +(C_y+C_{xy}B_x)(Y-B_y)+C_{xy}B_xB_y+C_xB_x+C_yB_y+C_0+v \\ &= C_{xy}(X-B_x)(Y-B_y)+C'_x(X-B_x)+C'_y(Y-B_y)+C'_0, \quad (1) \end{aligned}$$

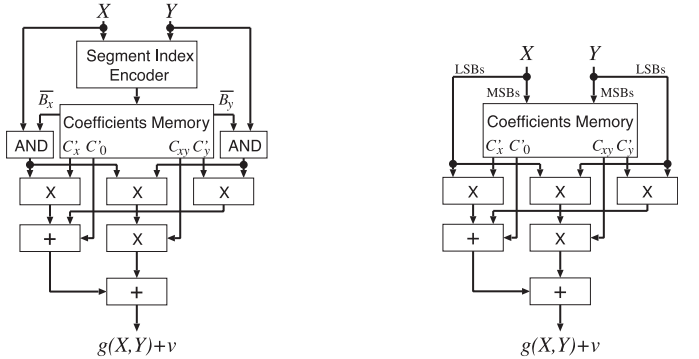
where $C'_x = C_x + C_{xy}B_y$, $C'_y = C_y + C_{xy}B_x$, and $C'_0 = C_{xy}B_xB_y + C_xB_x + C_yB_y + C_0 + v$.

4. Programmable Architectures for Two-Variable NFGs

4.1 Architectures Based on Recursive and Uniform Segmentations

Figure 3 shows two architectures for two-variable NFGs realizing (1). Figure 3 (a) and (b) show architectures based on recursive segmentation and uniform segmentation, respectively. The *segment index encoder* converts values of X and Y into a segment number. This, in turn, is applied as the address input of the *coefficients memory*. The coefficients are applied to adders and multipliers to form the polynomial value $g(X,Y)+v$. Note that Fig. 3 (a) uses bitwise AND gates to compute $X-B_x$ and $Y-B_y$. In recursive segmentation, we can realize $X-B_x$ and $Y-B_y$ using AND gates driven on one side by $\overline{B_x}$ and $\overline{B_y}$, respectively [15].

Note that Fig. 3 (b) has neither a segment index encoder nor bitwise AND gates. In uniform segmentation, the segment index encoder and bitwise AND gates are not necessary. This is because a segment number is obtained by the most significant bits of X and Y, and $X-B_x$ and $Y-B_y$, which are realized with bitwise AND gates in Fig. 3 (a), are obtained by the least significant bits.



(a) Architecture based on recursive segmentation. (b) Architecture based on uniform segmentation.

Fig. 3 Architectures for two-variable NFGs using bilinear interpolation.

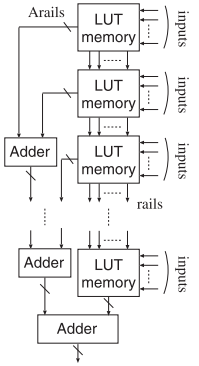
4.2 Architecture and Design Method for Segment Index Encoder

The segment index encoder realizes the segment index function: $\{0,1\}^n \times \{0,1\}^n \to \{0,1,\ldots,k-1\}$ shown in **Fig. 4** (a), where X and Y have n bits, and k denotes the number of segments. We realize this function with the architecture shown in Fig. 4 (b). In this architecture, the values of interconnecting lines between adjacent LUT memories represent sub-functions in the EVBDD (labeled *rails*), and the outputs from each LUT memory to the adders tally the function value (labeled *Arails*). Consider the design of the LUT cascade and adders in Fig. 4 (b), given the segmentation produced in Fig. 2.

We begin by representing the segment index function using an MTBDD. **Figure 5** illustrates the relationship between recursive segmentation and MTBDDs.

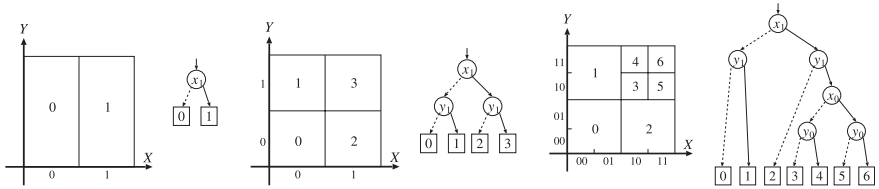
Segments	Index
$X_b \le X < P_0$ $Y_b \le Y < Q_0$	0
$X_b \le X < P_0$ $Q_0 \le X < Q_1$	1
⋮	⋮
$P_{r-1} \le X < Y_e$ $Q_{r-1} \le X < Y_e$	$k-1$

(a) Segment index function.



(b) LUT cascade and adders [15].

Fig. 4 Segment index encoder.



(a) Two segments. (b) Four segments. (c) Seven segments.

Fig. 5 Relationship between recursive segmentation and MTBDDs.

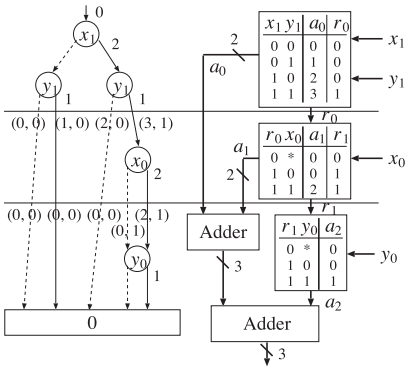


Fig. 6 Decomposition of the EVBDD.

Then, we convert the MTBDD into an EVBDD. By decomposing the EVBDD, as shown in **Fig. 6**, we obtain the architecture in Fig. 4 (b). In Fig. 6, the column labeled as 'r_i' in the table of each LUT memory denotes the rails that represent sub-functions in the EVBDD. And, the column 'a_i' in Fig. 6 denotes the Arails that represent the sum of weights of edges. In the EVBDD, "(a_i, r_i)" assigned to edges that cut across the horizontal lines represents the sum of weights and sub-functions, respectively. For more detail on this architecture, see 15).

In this architecture, the size of LUT memories realizing the recursive segmentation depends on the number of segments. Specifically,

Theorem 1 *Let $seg\_func(X, Y)$ be a segment index function obtained by a recursive planar segmentation. The segment index function can be realized by the segment index encoder shown in Fig. 4 (b) with at most $\lceil \log_2 k \rceil$ rails and at most $\lceil \log_2 k \rceil$ Arails, where k is the number of segments.*

Proof: See Appendix.

The segment index encoder satisfying Theorem 1 is obtained when the variable order for the EVBDD is $x_{l-1}, y_{l-1}, x_{l-2}, y_{l-2}, \ldots, x_{-m}, y_{-m}$ from the top to the bottom (see the proof in Appendix). We can also use the optimization techniques for multi-valued decision diagrams presented in 13) to optimize the variable order for EVBDD.

In our architectures, the coefficients memory and the LUT memories of the segment index encoder are implemented by RAMs. Thus, by changing the data for the coefficients memory and the LUT memories, a wide class of two-variable functions can be realized by a single architecture.

5. Design Method for Symmetric Functions

Definition 8 *A two-variable function $f(X, Y)$ is **symmetric** if $f(X, Y) = f(Y, X)$.*

Symmetric functions are commonly found in practical applications of NFGs. For example, $\sqrt{X^2 + Y^2}$, which is used in converting from rectangular to polar coordinates, is symmetric. This section presents an architecture and a design method taking advantage of the function's symmetry.

Definition 9 *A segmentation is **symmetric** if for every segment $\{[B_{x1}, E_{x1}), [B_{y1}, E_{y1})\}$ such that $B_{x1} \neq B_{y1}$ or $E_{x1} \neq E_{y1}$, there is another segment $\{[B_{x2}, E_{x2}), [B_{y2}, E_{y2})\}$ such that $B_{x1} = B_{y2}$, $E_{x1} = E_{y2}$, $B_{y1} = B_{x2}$, and $E_{y1} = E_{x2}$. **Symmetric segments** are a pair of such segments.*

Lemma 1 *Let $f(X, Y)$ be a symmetric function, and let $g_1(X, Y)$ and $g_2(X, Y)$ be bilinear interpolations of $f(X, Y)$ for symmetric segments. Then, $g_1(X, Y) = g_2(Y, X)$.*

Proof: See Appendix.

Theorem 2 *The segmentation of a symmetric function produced by the recursive planar segmentation algorithm is symmetric.*

Proof: See Appendix.

From Lemma 1 and Theorem 2, we can use only one bilinear interpolation to approximate the given symmetric function in symmetric segments. By assigning the same segment index to symmetric segments, we can reduce the size of the coefficients memory by nearly half.

Table 1 Number of segments for two segmentation methods.

No.	Function $f(X,Y)$	Domain		X and Y have 8-bit accuracy (Acceptable approx. error: 2^{-10})							X and Y have 12-bit accuracy (Acceptable approx. error: 2^{-14})						
		X	Y	Number of segments			R_{s1}	R_{s2}	Time [sec.]		Number of segments			R_{s1}	R_{s2}	Time [sec.]	
				Uni.	Recur.	Sym.	[%]	[%]	Uni.	Recur.	Uni.	Recur.	Sym.	[%]	[%]	Uni.	Recur.
f_0	$\sin(\pi X)\sqrt{Y}$	$[0,1)$	$(0,1)$	16,384	997	N/A	6	N/A	0.69	0.06	16,773,120	29,875	N/A	0.2	N/A	9.26	1.97
f_1	$\sin(\pi XY)$	$[0,1)$	$[0,1)$	1,024	508	263	50	26	0.07	0.03	16,384	8,389	4,232	51	26	1.00	0.42
f_2	X^4Y^5	$[0,1)$	$[0,1)$	4,096	193	N/A	5	N/A	0.30	0.02	65,536	3,592	N/A	5	N/A	4.73	0.33
f_3	$1/\sqrt{X^2+Y^2}$	$(0,1)$	$(0,1)$	65,025	2,344	1,195	4	2	0.02	0.07	16,769,025	103,046	51,687	0.6	0.3	6.10	3.91
f_4	$XY/\sqrt{X^2+Y^2}$	$(0,1)$	$(0,1)$	4,096	256	139	6	3	0.12	0.01	1,048,576	4,114	2,104	0.4	0.2	28.05	0.16
f_5	$WaveRings$	$[0,\pi]$	$[0,\pi]$	10,201	949	490	9	5	0.85	0.04	646,416	16,278	8,202	3	1	24.51	0.76
f_6	$Sombrero$	$(0,8)$	$(0,8)$	4,096	1,180	607	29	15	0.21	0.06	65,536	18,664	9,398	28	14	3.40	0.93
f_7	$\sqrt{X^2+Y^2}$	$(0,1)$	$(0,1)$	4,096	226	121	6	3	0.17	0.01	1,048,576	4,093	2,083	0.4	0.2	40.58	0.22
f_8	$\sqrt[3]{X^3+Y^3}$	$(0,1)$	$(0,1)$	4,096	232	127	6	3	0.33	0.02	1,048,576	3,955	2,027	0.4	0.2	78.21	0.41

Uni.: Uniform segmentation. Recur.: Recursive segmentation. R_{s1}: (No. of segments in Recur.) / (No. of segments in Uni.) × 100 (%).
Sym.: Symmetric segments are counted as one segment. R_{s2}: (No. of segments in Sym.) / (No. of segments in Uni.) × 100 (%).
Experiment environment: Sub Blade 2500 (Silver), UltraSPARC-IIIi 1.6 GHz, 6 GB memory, Solaris 9.

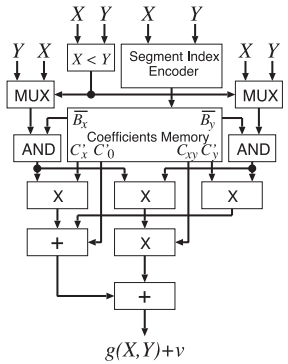


Fig. 7 Architecture for two-variable NFGs for symmetric functions.

Figure 7 shows an architecture for symmetric functions. Here, the coefficients memory stores only data for segments such that $X \leq Y$. For other segments, approximated values are computed using Lemma 1. Since the comparator and multiplexers operate in parallel with the segment index encoder, there is no speed penalty due to these additional circuits.

6. Experimental Results

6.1 Number of Segments and Computation Time for Algorithms

Table 1 shows the number of segments produced by the two segmentation algorithms presented in Section 3, and their computation time for various functions [1]. This table also shows the number of symmetric segments for symmetric functions. In this table, $WaveRings$ and $Sombrero$ are

$$WaveRings = \frac{\cos\left(\sqrt{X^2+Y^2}\right)}{\sqrt{X^2+Y^2+0.25}} \qquad Sombrero = \frac{\sin\left(\sqrt{X^2+Y^2}\right)}{\sqrt{X^2+Y^2}}$$

Table 1 shows that, for all functions except $\sin(\pi XY)$ and $Sombrero$, the recursive segmentation algorithm produces many fewer segments than the uniform segmentation algorithm. Especially, for higher accuracy, the number of segments needed in recursive segmentation is only a few percent of the number of segments needed in uniform segmentation. And, the number of symmetric segments is even smaller. Thus, the recursive segmentation algorithm and the symmetric technique significantly reduce the number of words in the coefficients memory. For $\sin(\pi XY)$ and $Sombrero$, the additional segments needed in uniform segmentation are not so large even for higher accuracy. This means that, for these

Table 2 Total memory sizes needed for the proposed NFGs.

No.	8-bit accuracy NFGs					12-bit accuracy NFGs				
	Uniform	Recursive	Sym.	R_{m1}	R_{m2}	Uniform	Recursive	Sym.	R_{m1}	R_{m2}
f_0	409,600	57,732	N/A	14	N/A	201,277,440	2,167,788	N/A	1	N/A
f_1	37,888	34,580	19,417	91	51	737,280	701,311	356,164	95	48
f_2	118,784	13,817	N/A	12	N/A	2,621,440	226,644	N/A	9	N/A
f_3	1,040,400	175,760	97,236	17	9	402,456,600	9,412,758	4,698,276	2	1
f_4	118,784	16,064	10,145	14	9	34,603,008	293,330	153,176	0.8	0.4
f_5	397,839	71,981	39,986	18	10	27,149,472	1,559,560	797,279	6	3
f_6	143,360	74,896	40,772	52	28	2,818,048	1,487,068	757,238	53	27
f_7	118,784	14,908	9,334	13	8	34,603,008	287,868	153,291	0.8	0.4
f_8	135,168	15,512	9,658	11	7	38,797,312	294,328	154,309	0.8	0.4

R_{m1}: Recursive / Uniform × 100 (%). R_{m2}: Sym. / Uniform × 100 (%).

functions, the uniform segmentation method also produces NFGs with reasonable size.

In addition, Table 1 shows that both algorithms produce segments with small CPU time. Such quick segmentation is useful to reduce design time for NFGs.

6.2 Memory Sizes Needed for Numeric Function Generators

Table 2 compares total memory sizes needed for the three NFGs shown in Fig. 3 and Fig. 7. Note that the NFGs based on recursive segmentation have two kinds of memories: coefficients memory and LUT memory. The memory size shown is the sum of the coefficients memory size and the LUT memory sizes.

Table 2 shows that, for all functions, NFGs based on recursive segmentation require smaller memory size than NFGs based on uniform segmentation, even though NFGs based on recursive segmentation have a segment index encoder. For example, for $f_4(X,Y) = XY/\sqrt{X^2+Y^2}$, the 12-bit accuracy NFG using recursive segmentation requires only 0.8% of memory required by uniform segmentation. Especially for symmetric functions, using the symmetric technique shown in Section 5 reduces the memory size significantly.

To understand the relation between memory size and accuracy, we designed NFGs for $XY/\sqrt{X^2+Y^2}$ with various accuracies. **Figure 8** plots memory sizes of the NFGs for 4 to 16-bit accuracies. There are four curves:

(1) a single look-up table in which the values assigned to X and Y form an address and the contents of that address is $f(X,Y)$,
(2) NFGs using uniform segmentation,

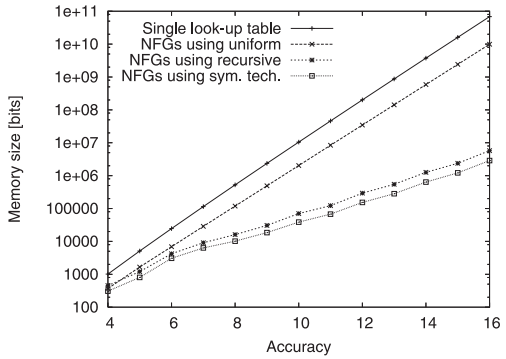


Fig. 8 Memory size versus accuracy for $XY/\sqrt{X^2+Y^2}$.

(3) NFGs using recursive non-uniform segmentation, and
(4) NFGs using the symmetric technique.

Interestingly, for this function, the memory size of the NFGs using uniform segmentation increases in the same way as the memory size of a single look-up table. On the other hand, the memory sizes of the NFGs using recursive segmentation and the NFGs using symmetric technique increase much more slowly than the other two. For 16-bit accuracy, the memory sizes of the NFG using recursive segmentation and the NFG using symmetric technique are only 0.06% and 0.03% of that of the NFG using uniform segmentation, respectively.

6.3 FPGA Implementation Results

To show the merits and demerits of the three proposed methods, we compare the performance of the NFGs designed by the three methods. We implemented 12-bit accuracy NFGs using the Altera Stratix III FPGA. Since the FPGA has *adaptive look-up tables (ALUTs)* that can realize fast adders, synchronous memory blocks, and dedicated multipliers, our NFGs are efficiently implemented by those hardware resources in the FPGA. **Table 3** compares the FPGA implementation results of the NFGs. In this table, the three columns labeled "Delay" show the total delay time of each NFG from the input to the output, in nanoseconds.

The NFGs based on uniform segmentation require fewer pipeline stages and have shorter delay than the recursive segmentation because they have no segment index encoder. However, for six functions, the memory needed for NFGs based

Table 3 FPGA implementation of 12-bit accuracy NFGs.

FPGA device: Altera Stratix III (EP3SL340F1517C2)						Logic synthesis tool: Altera QuartusII 9.0									
No.	Uniform segmentation					Recursive segmentation					Symmetric method				
	#ALUTs	#DSPs	Freq. [MHz]	#stages	Delay [ns]	#ALUTs	#DSPs	Freq. [MHz]	#stages	Delay [ns]	#ALUTs	#DSPs	Freq. [MHz]	#stages	Delay [ns]
f_0	–	0	–	1	–	440	6	230	15	65	N/A	N/A	N/A	N/A	N/A
f_1	49	5	203	4	20	271	10	191	9	47	297	9	191	10	52
f_2	206	0	306	4	13	266	8	187	10	53	N/A	N/A	N/A	N/A	N/A
f_3	–	0	–	1	–	–	7	–	18	–	644	7	174	16	92
f_4	–	0	–	4	–	220	6	228	10	44	273	6	222	11	50
f_5	–	3	–	4	–	477	10	221	13	59	493	10	221	13	59
f_6	153	4	230	4	17	336	8	192	11	57	293	8	191	11	57
f_7	–	1	–	4	–	237	6	231	10	43	279	6	231	11	48
f_8	–	1	–	4	–	236	8	199	10	50	255	8	199	10	50

–: NFGs cannot be mapped into the FPGA due to insufficient memory blocks.
#ALUTs: Number of ALUTs. #DSPs: Number of 18-bit × 18-bit DSP units. Freq.: Operating frequency. #stages: Number of pipeline stages.

on uniform segmentation is so large that they could not be implemented on the FPGA. Note that NFGs that have only one pipeline stage in Table 3 are realized with a single look-up table due to the excessively many segments. On the other hand, for all functions except for $f_3(X,Y) = 1/\sqrt{X^2+Y^2}$, the NFGs based on recursive segmentation do not require excessive memory size and can be implemented on the FPGA. Further, the successful implementations achieve a high operating frequency. Since the symmetric technique significantly reduces memory size, even function f_3 can be implemented with the FPGA. But, the symmetric technique has some speed penalty because it produces a slightly more complex segment index encoder.

In this way, the three methods have different merits and demerits, and thus, we can use the three methods appropriately depending on applications and numeric functions.

Although the recursive segmentation and symmetric technique have some speed penalty as shown in Table 3, the penalty is reasonable. To show that, we compare our NFGs with an NFG designed in a conventional (trivial) manner that uses a combination of one-variable NFGs and basic operations like addition and multiplication. We implemented $f_4(X,Y) = XY/\sqrt{X^2+Y^2}$ with the same FPGA using a one-variable NFG for $1/\sqrt{X}$, two squaring circuits, an adder, and two multipliers. The one-variable NFG was realized by the method shown in 15),

Table 4 FPGA implementation of various NFGs for $XY/\sqrt{X^2+Y^2}$.

FPGA device:	Altera Stratix III (EP3SL340F1517C2)					
Logic synthesis tool:	Altera QuartusII 9.0					
NFGs	Memory [bits]	#LEs	#DSPs	Freq. [MHz]	#stages	Delay [nsec.]
	12-bit accuracy					
One-variable	269,136	266	10	192	14	73
Uniform	34,603,008	–	0	–	4	–
Recursive	293,330	220	6	228	10	44
Symmetric	153,176	273	6	222	11	50

which is based on linear approximation and non-uniform segmentation. **Table 4** compares the results with our NFGs.

Our NFG based on recursive segmentation requires fewer ALUTs and DSPs than the implementation using one-variable NFG, and has much shorter delay. Especially, the NFG designed by the symmetric method achieves both less memory and shorter delay. This shows that the speed penalties caused by the recursive segmentation and the symmetric method are small enough.

From these results, we can see that by designing two-variable functions using one-variable NFGs, the required memory size can be reduced significantly. However, depending on the functions, it can produce a slow implementation because of additional logic, such as multipliers. Also, complicated architectures using

one-variable NFGs make error analysis harder, and it is harder to guarantee output accuracy. This increases design time. On the other hand, for a large class of functions, we can automatically generate fast and compact NFGs whose output accuracy is guaranteed.

7. Concluding Remarks

We have proposed programmable architectures and design methods for numeric function generators of two-variable functions. To realize a two-variable function in hardware, we partition the given domain of the function into segments, and approximate the given function by a polynomial in each segment. In this paper, we presented two planar segmentation algorithms which partition a given domain of two-variable function efficiently. We also presented a design method for symmetric two-variable functions. To the best of our knowledge, these are the first systematic design methods based on piecewise polynomial approximation for two-variable functions. Experimental results showed that for a complicated function, our automatically generated NFG achieves higher performance than the NFG that is manually designed in a conventional manner.

In the proposed architectures, the coefficients memory and the LUT memories of the segment index encoder can be implemented by embedded RAMs in an FPGA (e.g., M4Ks in Altera FPGAs). Thus, by changing the data for the coefficients memory and the LUT memories, a wide class of two-variable functions can be realized by a single architecture. Since just changing the RAM data can switch functions, we can switch functions without reprogramming the FPGA.

The algorithms and architectures presented in this paper can be easily extended to functions with three or more variables.

Acknowledgments This research is partly supported by the Grant in Aid for Scientific Research of the Japan Society for the Promotion of Science (JSPS), Knowledge Cluster Initiative (the second stage) of MEXT (Ministry of Education, Culture, Sports, Science and Technology), a contract with the National Security Agency, and the MEXT Grant-in-Aid for Young Scientists (B), 20700051, 2009.

References

1) Anton, H.: *Multivariable Calculus*, John Wiley & Sons, Inc. (1995).
2) Bryant, R.E.: Graph-based algorithms for boolean function manipulation, *IEEE Trans. Comput.*, Vol.C-35, No.8, pp.677–691 (1986).
3) Clarke, E.M., McMillan, K.L., Zhao, X., Fujita, M. and Yang, J.: Spectral transforms for large Boolean functions with applications to technology mapping, *Proc. 30th ACM/IEEE Design Automation Conference*, pp.54–60 (June 1993).
4) De Caro, D. and Strollo, A.G.M.: High-performance direct digital frequency synthesizers using piecewise-polynomial approximation, *IEEE Trans. Circuit and Systems*, Vol.52, No.2, pp.324–337 (2005).
5) Detrey, J. and de Dinechin, F.: Table-based polynomials for fast hardware function evaluation, *16th IEEE Inter. Conf. Application-Specific Systems, Architectures, and Processors* (*ASAP'05*), pp.328–333 (2005).
6) Gutierrez, R. and Valls, J.: Implementation on FPGA of a LUT based $atan(y/x)$ operator suitable for synchronization algorithms, *Proc. IEEE Conf. Field Programmable Logic and Applications*, pp.472–475 (Aug. 2007).
7) Huang, Z. and Ercegovac, M.D.: FPGA implementation of pipelined on-line scheme for 3-D vector normalization, *Proc. 9th Annual IEEE Symp. Field-Programmable Custom Computing Machines* (*FCCM'01*), pp.61–70 (Apr. 2001).
8) Lai, Y-T. and Sastry, S.: Edge-valued binary decision diagrams for multi-level hierarchical verification, *Proc. 29th ACM/IEEE Design Automation Conference*, pp.608–613 (1992).
9) Lai, Y-T., Pedram, M. and Vrudhula, S.B.: EVBDD-based algorithms for linear integer programming, spectral transformation and functional decomposition, *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, Vol.13, No.8, pp.959–975 (1994).
10) Lee, D.-U., Luk, W., Villasenor, J. and Cheung, P.Y.K.: Hierarchical segmentation schemes for function evaluation, *Proc. IEEE Conf. Field-Programmable Technology*, Tokyo, Japan, pp.92–99 (Dec. 2003).
11) Meinel, C. and Theobald, T.: *Algorithms and Data Structures in VLSI Design: OBDD — Foundations and Applications*, Springer (1998).
12) Muller, J.-M.: *Elementary Function: Algorithms and Implementation*, Birkhauser Boston, Inc., New York, NY, second edition (2006).
13) Nagayama, S. and Sasao, T.: On the optimization of heterogeneous MDDs, *IEEE Trans. CAD*, Vol.24, No.11, pp.1645–1659 (2005).
14) Nagayama, S., Sasao, T. and Butler, J.T.: Compact numerical function generators based on quadratic approximation: Architecture and synthesis method, *IEICE Trans. Fundamentals*, Vol.E89-A, No.12, pp.3510–3518 (2006).
15) Nagayama, S., Sasao, T. and Butler, J.T.: Design method for numerical function generators using recursive segmentation and EVBDDs, *IEICE Trans. Fundamentals*, Vol.E90-A, No.12, pp.2752–2761 (2007).
16) Nagayama, S., Butler, J.T. and Sasao, T.: Programmable numerical function generators for two-variable functions, *EUROMICRO Conference on Digital System Design* (*DSD-2008*), pp.891–898 (Sep. 2008).

17) Nagayama, S., Sasao, T. and Butler, J.T.: Numerical function generators using bilinear interpolation, *Proc. IEEE International Conference on Field Programmable Logic and Applications*, pp.463–466 (Sep. 2008).
18) Piñeiro, J.-A., Oberman, S.F., Muller, J.-M. and Bruguera, J.D.: High-speed function approximation using a minimax quadratic interpolator, *IEEE Trans. Comp.*, Vol.54, No.3, pp.304–318 (2005).
19) Sasao, T., Nagayama, S. and Butler, J.T.: Numerical function generators using LUT cascades, *IEEE Trans. Comput.*, Vol.56, No.6, pp.826–838 (2007).
20) Schulte, M.J. and Stine, J.E.: Approximating elementary functions with symmetric bipartite tables, *IEEE Trans. Comput.*, Vol.48, No.8, pp.842–847 (1999).
21) Späth, H.: *Two Dimensional Spline Interpolation Algorithms*, A K Peters, Ltd., Wellesley, MA (1995).
22) Takagi, N. and Kuwahara, S.: A VLSI algorithm for computing the Euclidean norm of a 3D vector, *IEEE Trans. Comput.*, Vol.49, No.10, pp.1074–1082 (2000).

Appendix

This appendix shows the proofs of Theorem 1, Lemma 1, and Theorem 2.

The proof of Theorem 1 is based on a theorem proven in Ref. 15). Specifically, it was shown that

Theorem A *Let $g(Z)$ be a k-valued monotone increasing function. The function $g(Z)$ can be realized by the segment index encoder shown in Fig. 4 (b) with at most $\lceil \log_2 k \rceil$ rails and $\lceil \log_2 k \rceil$ Arails*[15].

Theorem 1 *Let $seg\_func(X, Y)$ be a segment index function obtained by a recursive planar segmentation. The segment index function can be realized by the segment index encoder shown in Fig. 4 (b) with at most $\lceil \log_2 k \rceil$ rails and at most $\lceil \log_2 k \rceil$ Arails, where k is the number of segments.*

Proof: By forming a variable

$$Z = (x_{l-1}\ y_{l-1}\ x_{l-2}\ y_{l-2}\ \cdots\ x_{-m}\ y_{-m})$$

from X and Y, $seg\_func(X, Y)$ obtained by the recursive planar segmentation algorithm can be converted into a k-valued monotone increasing function $g(Z)$. Therefore, from Theorem A, we have this theorem. ∎

Lemma 1 *Let $f(X, Y)$ be a symmetric function, and let $g_1(X, Y)$ and $g_2(X, Y)$ be bilinear interpolations of $f(X, Y)$ for symmetric segments. Then, $g_1(X, Y) = g_2(Y, X)$.*

Proof: Let $g_1(X, Y) = C_{xy1}XY + C_{x1}X + C_{y1}Y + C_{01}$ and $g_2(X, Y) = C_{xy2}XY + C_{x2}X + C_{y2}Y + C_{02}$. From the definition of bilinear interpolation, in symmetric segments, the following hold: $C_{xy1} = C_{xy2}$, $C_{x1} = C_{y2}$, $C_{y1} = C_{x2}$, and $C_{01} = C_{02}$. Therefore, we have the lemma. ∎

To prove Theorem 2, we define the following:

Definition A ***Diagonal segment*** *$\{[B_x, E_x), [B_y, E_y)\}$ is a segment such that $B_x = B_y$ and $E_x = E_y$.*

Theorem 2 *The segmentation of a symmetric function produced by the recursive planar segmentation algorithm is symmetric.*

Proof: Let $\{[B_{x1}, E_{x1}), [B_{y1}, E_{y1})\}$ and $\{[B_{x2}, E_{x2}), [B_{y2}, E_{y2})\}$ be symmetric segments. Since $(B_{x1} + E_{x1})/2 = (B_{y2} + E_{y2})/2$ and $(B_{y1} + E_{y1})/2 = (B_{x2} + E_{x2})/2$, the segmentation of the symmetric segments into four equal-sized square segments is symmetric. The segmentation of diagonal segment into four equal-sized square segments is also symmetric.

From Lemma 1, the maximum approximation errors caused in symmetric segments are equal. Thus, if a segment is partitioned, then another segment symmetric to the segment is also partitioned.

Therefore, the recursive planar segmentation algorithm produces a symmetric segmentation for a symmetric function. ∎

(Received May 7, 2009)
(Revised September 1, 2009)
(Accepted October 31, 2009)
(Released February 15, 2010)

(Recommended by Associate Editor: *Yusuke Matsunaga*)

Shinobu Nagayama received the B.S. and M.E. degrees from Meiji University, Kanagawa, Japan, in 2000 and 2002, respectively, and the Ph.D. degree in computer science from Kyushu Institute of Technology, Iizuka, Japan, in 2004. He is now a lecturer at Hiroshima City University, Hiroshima, Japan. He received the Outstanding Contribution Paper Award from the IEEE Computer Society Technical Committee on Multiple-Valued Logic (MVL-TC) in 2005 for a paper presented at the International Symposium on Multiple-Valued Logic in 2004, and the Excellent Paper Award from the Information Processing Society of Japan (IPS) in 2006. His research interest includes numeric function generators, decision diagrams, software synthesis, and embedded systems.

Tsutomu Sasao received the B.E., M.E., and Ph.D. degrees in electronics engineering from Osaka University, Osaka, Japan, in 1972, 1974, and 1977, respectively. He has held faculty/research positions at Osaka University, Japan, the IBM T.J. Watson Research Center, Yorktown Heights, New York, and the Naval Postgraduate School, Monterey, California. He is now a Professor of the Department of Computer Science and Electronics at Kyushu Institute of Technology, Iizuka, Japan. His research areas include logic design and switching theory, representations of logic functions, and multiple-valued logic. He has published more than nine books on logic design, including *Logic Synthesis and Optimization, Representation of Discrete Functions, Switching Theory for Logic Synthesis*, and *Logic Synthesis and Verification*, Kluwer Academic Publishers, 1993, 1996, 1999, and 2001, respectively. He has served as Program Chairman for the IEEE International Symposium on Multiple-Valued Logic (ISMVL) many times. Also, he was the Symposium Chairman of the 28th ISMVL held in Fukuoka, Japan, in 1998. He received the NIWA Memorial Award in 1979, Distinctive Contribution Awards from the IEEE Computer Society MVL-TC for papers presented at ISMVLs in 1986, 1996, 2003 and 2004, and Takeda Techno-Entrepreneurship Award in 2001. He has served as an Associate Editor of the *IEEE Transactions on Computers*. He is a fellow of the IEEE.

Jon T. Butler received the B.E.E. and M.Engr. degrees from Rensselaer Polytechnic Institute, Troy, New York, in 1966 and 1967, respectively. He received the Ph.D. degree from The Ohio State University, Columbus, in 1973. Since 1987, he has been a professor at the Naval Postgraduate School, Monterey, California. From 1974 to 1987, he was at Northwestern University, Evanston, Illinois. During that time, he served two periods of leave at the Naval Postgraduate School, first as a National Research Council Senior Postdoctoral Associate (1980–1981) and second as the NAVALEX Chair Professor (1985–1987). He served one period of leave as a foreign visiting professor at Kyushu Institute of Technology, Iizuka, Japan. His research interests include logic optimization and multiple-valued logic. He has served on the editorial boards of the *IEEE Transactions on Computers*, *Computer*, and IEEE Computer Society Press. He has served as the editor-in-chief of *Computer* and IEEE Computer Society Press. He received the Award of Excellence, the Outstanding Contributed Paper Award, and a Distinctive Contributed Paper Award for papers presented at the International Symposium on Multiple-Valued Logic. He received the Distinguished Service Award, two Meritorious Awards, and nine Certificates of Appreciation for service to the IEEE Computer Society. He is a fellow of the IEEE.